Deep-Tree Implementation of the Multi-Resolution Approximation's Domain Decomposition

Math 540 - Final Project Paper

Lewis Blake

May 6, 2019

1 Introduction

1.1 The problem

Two primary objectives of spatial statistics are to perform parameter inference and spatial predictions. To that end, it is often desirable to model a spatial data set as a Gaussian Process (GP). A GP is defined by the property that any finite combination of observations follows a multivariate-normal distribution (MVN). For a random vector $\boldsymbol{X} = (X_1, X_2, ..., X_n)^T$, we say $\boldsymbol{X}$ follows a MVN distribution (symbolically, $\boldsymbol{X} \sim N_n(\boldsymbol{\mu}, \Sigma)$) if and only if the probability density function (pdf) of $\boldsymbol{X}$ is given by

$$f(\boldsymbol{X}) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{X}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{X}-\boldsymbol{\mu})\right).$$

It is clear that the MVN pdf depends on Σ^{-1} and $\det(\Sigma)$. Consequently, directly evaluating this pdf incurs $\mathcal{O}(n^2)$ memory and $\mathcal{O}(n^3)$ time complexity, which becomes computationally unfeasible when the number of observations, n, is on the order of 10^5 or more.

To overcome this issue, many alternatives to directly modeling spatial data as GP have been proposed in recent years. In this project, I focus on one such method: the Multi-Resolution Approximation (MRA) (1).

1.2 The approach

The MRA has been shown to be one of the most computationally efficient and accurate methods to analyze large spatial data sets (2). For a full description of the MRA's statistical foundations, please see (1). A brief overview of the MRA model, taken from §2 of (3), is presented for convenience below with permission from the author.

The spatial field of interest is modeled via basis function representation of a GP. The true spatial field is denoted $\{y_0(\mathbf{s}) : \mathbf{s} \in \mathcal{D}\}$, or $y_0(\cdot)$, on a continuous domain $\mathcal{D} \subset \mathbb{R}^d$, $d \in \mathbb{N}^+$. Typically we take $d = 2$ of $d = 3$. Assumed is that $y_0(\cdot) \sim GP(0, C_0)$ is a zero-mean Gaussian process with covariance function C_0, that is known up to a vector of parameters $\boldsymbol{\theta}$.

Once data has been observed at n spatial locations, by the GP assumption, the data follows a n-dimensional MVN distribution. Directly evaluating this distribution results in the computationally burdensome task of inverting and calculating the determinant of an $n \times n$ matrix. To facilitate computationally feasible approximations of the MVN distribution, the following simplifying assumptions are introduced to define the MRA.

To begin with, the MRA defines a recursive partitioning of the spatial domain $\mathcal{D}$, in which each of the J regions is further divided into J smaller regions, and this process is repeated M times. Let $\mathcal{D}_{j_1,...,j_{m-1}}$ be a region at level $m-1$. We can write $\mathcal{D}_{j_1,...,j_{m-1}}$ as a union of regions at one resolution finer (level m) as follows:

$$\mathcal{D}_{j_1,...,j_{m-1}} = \bigcup_{j_m=1,...,J} \mathcal{D}_{j_1,...,j_m}, \qquad j_1, ..., j_m = 1, ..., J; \qquad m = 1, ..., M.$$

In the MRA, for the GP $y_0(\cdot)$, $[y_0(\cdot)]_{[m]}$ is defined as a "block-independent" version of $y_0(\cdot)$ between regions at resolution m. That is, $[C_0]_{[m]}(\mathbf{s}_1, \mathbf{s}_2) = C_0(\mathbf{s}_1, \mathbf{s}_2)$ if $\mathbf{s}_1, \mathbf{s}_2$ are in the same region $\mathcal{D}_{j_1,...,j_m}$ and $[C_0]_{[m]}(\mathbf{s}_1, \mathbf{s}_2) = 0$ otherwise, where $C_0(\cdot, \cdot)$ is the covariance function. The recursive partitioning of the domain along with the block-independence assumption leads to a natural interpretation of domain partitioning inheriting a parent-child hierarchical structure. In explicit, at each level $m = 1, ..., M-1$, we can view each region $\mathcal{D}_{j_1,...,j_m}$ as the parent of the J subregions $\mathcal{D}_{j_1,...,j_{m+1}}$ at level $m+1$ contained within $\mathcal{D}_{j_1,...,j_m}$. Further, by block-independence, at each level the GP within each region $\mathcal{D}_{j_1,...,j_m}$ is only assumed statistically dependent on its parental hierarchy (i.e., all regions at coarser resolutions containing $\mathcal{D}_{j_1,...,j_m}$).

Also defined are a set of r knots (with $r << n$) at each resolution that all lie within a particular subregion $\mathcal{D}_{j_1,...,j_m}$. The knots are the locations at which the basis function attain their maximum. At the finest resolution, M, we define the knots to be the observations within that region. By placing knots within each subregion, instead of working with $n \times n$ matrices, we can reasonably well approximate $y_0(\cdot)$ by working mostly with $r \times r$ matrices in a computationally feasible manner. To model the spatial field as a GP, the MRA iteratively approximates $y_0(\cdot)$ and covariance function $C_0(\cdot, \cdot)$ at resolutions $m = 1, ..., M$ dependent on the knots and partitions. At coarser resolutions, the MRA captures large-distance spatial trends in $y_0(\cdot)$. By increasing number of levels used in the approximation, the MRA captures shorter range variability as well.

1.3 The contribution

The long term objective of this project is to implement a parallel "Deep-Tree MRA" in C++. Towards this goal, for my final project in this course I implemented the portion of the code responsible for building the hierarchical structure. Through future work outside the scope of this course, I will provide fast, efficient, and scalable software to analyze massive spatial data sets. In Summer 2018, I was a Graduate Research Assistant at the National Center for Atmospheric Research (NCAR). During this time, I developed a parallel implementation of the MRA and researched this implementation's computational performance in comparison to the serial implementation by conducting timing and memory profiling studies on the Cheyenne HPC. Studying the parallel MRA's computational performance prompted the necessity of developing a new parallelization scheme which would scale to much larger data sets, as memory constraints limited the feasible data size on a single node to approximately 2.7 million observations (3).

My collaborators at NCAR and I designed two new MRA approaches in MATLAB and C++. I implemented a one of these approaches, called the "Shallow-Tree" MRA, using codistributed arrays in MATLAB as part of an independent study over the Fall 2018 semester. The key strategy of this implementation is to distribute computations in parallel across nodes in order to overcome present memory bounds. While this new Shallow-Tree MRA is still in its infancy, initial results are promising. Through code object distribution, the feasible scalability of the data size has increased to around 47 million observations. The MATLAB implementation can evaluate a likelihood with this many observations in 1194.16 seconds with a standard deviation of 4.90 seconds and perform spatial predictions in 1580.66 seconds with a standard deviation of 11.50 seconds.

This new scheme has also led to a reduction in runtimes for smaller data sets around 3 million observations. There is still room for improvement, which is the focus of this project. By efficiently implementing the a slightly modified version of this approach in C++, I hope to gain further efficiency and be able to tackle data sets on the order to 50 million or even 100 million observations while reducing execution times.

2 Model Implementation: The Deep-Tree Approach

There are many similarities between the Shallow-Tree MRA I implemented in MATLAB and the beginnings of the Deep-Tree MRA I implemented for my final project in this course. The goal of both the Shallow-Tree and Deep-Tree approach is to break up the problem across workers such that on no node is the available memory exceeded. Another objective of this approach is that memory and execution time are not spent on communication between nodes.

The MRA progresses from coarsest resolution to finest resolution creating the "prior distribution", and then from finest resolution to coarsest resolution calculating the "posterior distribution". Further by block-independence, versions of the GP at each level are assumed to be statistically independent across regions. A consequence of these modeling assumptions is that it allows breaking the problem apart into smaller

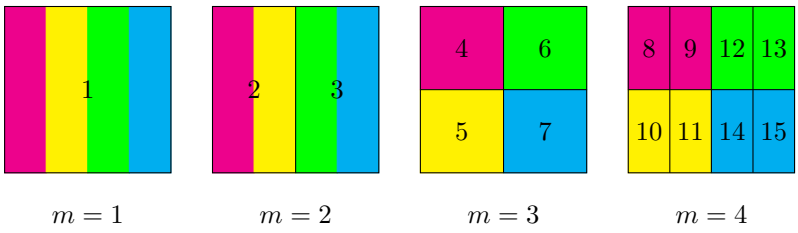

Figure 1: Toy example of the MRA domain decomposition and how the Deep-Tree MRA breaks up regions across cores. At level $m = 3$, each region is assigned to a core. Consequently, regions at finer resolutions (i.e., larger values of m) contained within a given region are assigned to that core as well as all regions coarser resolutions (i.e., smaller values of m) that contain that region.

pieces that can be calculated separately in parallel and then "stitched" back together later in a statistically justifiable manner. The pieces of the problem that are dealt with independently in parallel are done so by workers.

Conceptually, both the MATLAB Shallow-Tree MRA and the C++ Deep-Tree MRA split the number of regions at a chosen level across cores/workers. Borrowing the naming conventions from the MATLAB Shallow-Tree implementation, this number of levels is called `NUM_LEVEL_BEGIN_PARALLEL` or P for shorthand. We also denote `NUM_LEVELS_SERIAL_S`, or S, as level $P-1$. At level $m = P$, each core may have one or more $\mathcal{D}_{j_1,...,j_P}$ assigned to it. We call the set of regions at coarser resolutions (i.e., $m = 1, ..., S$) a region's *ancestry* and the regions contained within $\mathcal{D}_{j_1,...,j_P}$ its *descendants*. See Figure 1 for a visual representation.

It is important to note that while in the MATLAB Shallow-Tree implementation, levels $m = 1, ..., S$ are calculated in serial by the `master` core, the C++ Deep-Tree implementation does something slightly different. Rather than just having the `master` core compute the first few levels in serial and them sending the associated data structures to workers with perhaps `MPI_Scatter` or with `MPI_Send` and `MPI_Recv`, I reduce the communication between workers and the `master` by having each core perform calculations for the entirety of the ancestry and descendants for each region $\mathcal{D}_{j_1,...,j_P}$ assigned to it at level $m = P$. By doing so, I reduce communication, the memory overhead is (theoretically) identical to sending the data structures to the workers, and no (theoretical) additional execution time is incurred as the workers would have been idle during the `master`'s serial computations regardless. This difference between having the just the `master` compute levels $m = 1, ..., S$ or having all cores compute levels $m = 1, ..., S$ is the primary distinction between the Shallow-Tree and Deep-Tree implementations respectively.

2.1 The code

In an effort to minimize runtime spent on communication between workers and memory overhead on any particular node, we design our Deep-Tree implementation using the `MPI` library. The user may specify parameters within the `userInput.txt` file. Each input is put on a new line within this file and the program then reads each line to assign it to a variable within the code base. A benefit of this approach is that the code only needs to be compiled once, and then different data sets with different parameters settings can easily be used for computations. The structure on the `userInput.txt` file is as follows:

```
dataSource
computationType
NUM_LEVELS_M
NUM_PARTITIONS_J
NUM_KNOTS_r
offsetPercentage
NUM_LEVELS_SERIAL_S
```

The first line within `userInput.txt` is a string that corresponds to a data set within the `Data` folder. Three data sets are inlucude: `amsrDay.bin`, `modiDay.bin`, and `satelliteData.bin` contained within the

`Data` directory. These files contain (x, y, z) coordinates for spatial data sets on the order of $3,000,000$, $47,000,000$, and $100,000$ respectively. For testing purposes, the default is set as the small satellite data with $M = 9$, $J = 2$, $r = 64$, `offsetPercentage` $= 0.01$, and $S = 5$.

It is important to note that the number of cores utilized in any execution is limited by the number of regions at the level at which regions are split across cores. More over, the number of cores used must also be a power of J. For example, in Figure 1, regions are split across workers at level $m = 3$. That is, we have chosen $S = 2$ in the `userInput.txt` file. In this case, either 2 or 4 cores can be commissioned to run the code. This ensures a good load balance across cores. In general, one can not set `numCores` $>$ `nRegions(P)`, where `nRegions` returns the number of regions at a given level.

There are four main files responsible for building the hierarchical structure: `main.cpp`, `functions.cpp`, `functions.h`, and `buildStructure.cpp`. The main file is responsible for calling smaller functions and major model routines. The file `functions.cpp` is where many of the smaller model functions are implemented and its associated header file is `functions.h`. The file `buildStructure.cpp` is where the core functionality of building the hierarchical structure is implemented, and therefore is given its own file. The output of the model are progress indicators and timing results, however I am less interested in the outputs of the model at this point, and rather the focus is on setting up data structures optimally for future work. In particular, the outputs from `buildStructure.cpp`; `knots`, `outputData`, and `partitions` are stored specific to the regions assigned to each core.

Presently, the `MPI` commands used in the mode are `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Abort`, and `MPI_Finalize`. Such few commands are required to build the hierarchical structure because I have designed it so be embarrassingly parallel. When implementing the statistical inference, particularly the posterior inference, synchronization and communication between cores will be needed.

3 Timing Study

Timing studies with the `satelliteData.bin` and `modisDay.bin` data are performed to analyze performance on both "small" and "large" data sets. For the remainder of the study I will refer to the `satelliteData.bin` as the "Small Data" and `modisDay.bin` as the "Large Data".

3.1 Small Data Timing Study

Throughout I set $M = 9$, $S = 5$, $r = 64$, and $J = 2$. All tests using ≤ 16 cores are run on a single node whereas the test with 32 cores is run on two nodes. Timings are given in the Table 2 and Figure 3 provides visual representations for the accrued results and various metrics.

Number of Cores	Execution Time (seconds)
1	1.5956e-01
2	4.6421e-02
4	1.4865e-02
8	1.7356e-02
16	1.7331e-02
32	1.7579e-02

Figure 2: Timing results for the small data set. Only the case with 32 cores required computing on two nodes. All other tests were executed on one node.

3.2 Large Data Timing Study

Throughout I set $M = 18$, $S = 6$, $r = 64$, and $J = 2$. All tests where run with one core on any given number of nodes. For example a test, with 16 cores was run over 16 nodes with 1 core on each node. This was done because the problem is memory bound and the data set is quite large. Moreover, the smallest number of cores that could facilitate computations was 4 cores, so we look at relative improvements in performance. Timings are given in the Table 4 and Figure 3.2 provides visual representations for the accrued results and various metrics.

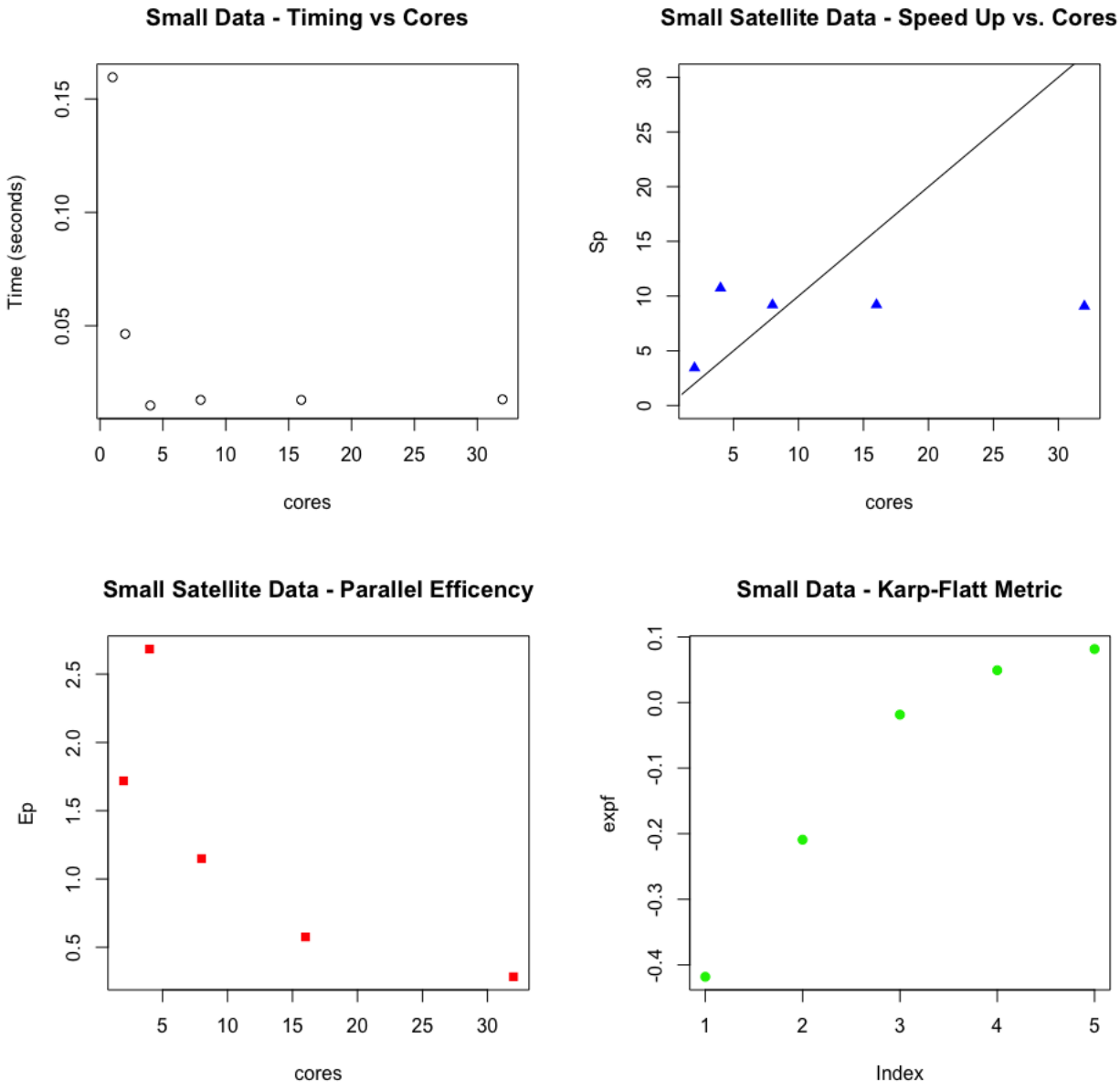

Figure 3: Timing study results for the small data set.

Number of Cores	Execution Time (seconds)
4	2.0696e+03
8	4.6819e+02
16	1.9023e+02
32	3.2042e+01
64	3.9257e+01

Figure 4: Timing results for the large data set. For each set up, one MPI process was used on each core.

4 Conclusions and Future Work

The timing study shows that super-linear run times can be achieved for certain model configurations and a given n. The optimal case for the Small data set was with 4 cores where each core was responsible for 8 regions

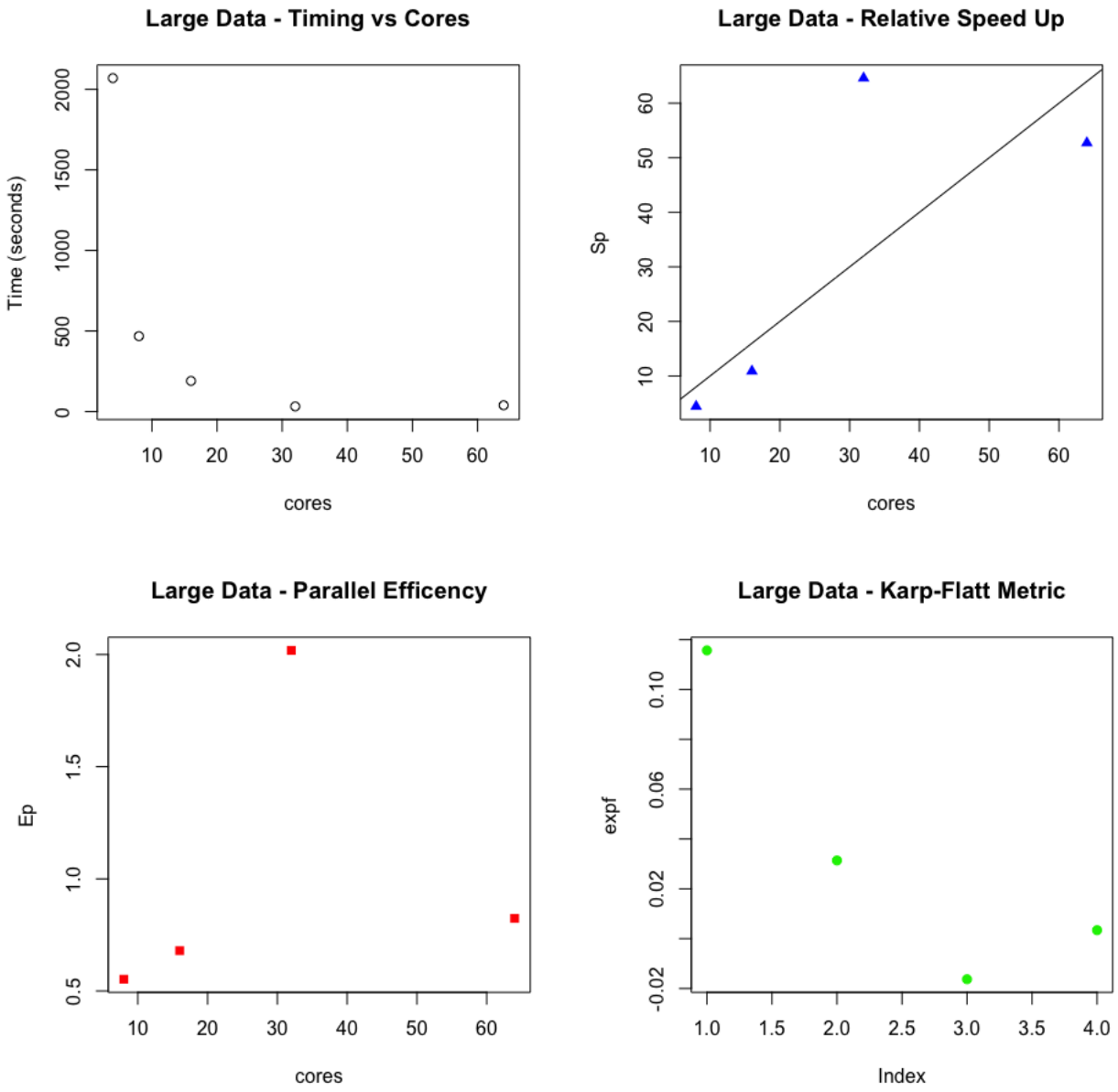

Figure 5: Timing results for the large data set.

at `NUM_LEVEL_BEGIN_PARALLEL`. The optimal case for the large data set was with 32 cores where each core was responsible for 2 regions at `NUM_LEVEL_BEGIN_PARALLEL`. Where super-linear speed-ups are observed, the Karp-Flatt Metric estimate of the inherently serial fraction of the code is estimated to be negative because $1/S_p < 1/P$, and hence is not applicable in these circumstances. For most parameter settings there are sub-linear timing results, which is more aligned to expectation. These results are promising for the Deep-Tree implementation and its ability to scale to very large geospatial data sets. Future work will take the current model implementation, reconsider how observations are assigned to regions at the finest resolution, and implement the statistical inference.

References

[1] M. Katzfuss, "A multi-resolution approximation for massive spatial datasets," *Journal of the American Statistical Association*, vol. 117, no. 517, pp. 201–214, 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/01621459.2015.1123632

[2] M. J. Heaton, A. Datta, A. O. Finley, R. Furrer, J. Guinness, R. Guhaniyogi, F. Gerber, R. B. Gramacy, D. Hammerling, M. Katzfuss, F. Lindgren, D. W. Nychka, F. Sun, and A. Zammit-Mangion, "A case study competition among methods for analyzing large spatial data," *Journal of Agricultural, Biological and Environmental Statistics*, Dec 2018. [Online]. Available: https://doi.org/10.1007/s13253-018-00348-w

[3] L. R. Blake, P. Simonson, and D. Hammerling, "Parallel implementation and computational analysis of the multi-resolution approximation," *NCAR Technical Note*, 2018. [Online]. Available: http://n2t.net/ark:/85065/d7k07756